Recap: Model Checking for model-based testing

From test spec.

- **LTL Property**
- **System Model**
- **Model Checking**
- Yes, with Counter-example trace
- No, with Counter-example trace

Encoding test specifications

- **Def. 1**
  - A trace $\sigma$ satisfies a test specification $M$ if $\sigma$ contains at least one linearization of $M$ as a contiguous subsequence.
  - Given MSC $M$,
    - define $\text{Lin}(M)$ = set of linearizations of $M$.
    - For each linearization $\sigma = e_1 e_2 \ldots e_k$ define
      - $\text{prop}_\sigma = \Box_{e_1}(X_{e_2}(X_{e_3}(\ldots(X_{e_k}(\ldots)))$)
      - Define property $\phi_M$ corresponding to $M$ as
        - $\phi_M = \neg \bigvee_{\sigma \in \text{Lin}(M)} \text{prop}_\sigma$
    - A counter-example to $\phi_M$ is a test satisfying $M$.

Encoding test specifications

- **Def. 2**
  - A trace $\sigma$ satisfies a test specification $M$ if $\sigma$ contains at least one linearization of $M$ as a subsequence.
  - Given MSC $M$,
    - define $\text{Lin}(M)$ = set of linearizations of $M$.
    - For each linearization $\sigma = e_1 e_2 \ldots e_k$ define
      - $\text{prop}_\sigma = \neg (e_1 \lor e_2 \lor \ldots \lor e_k)$
      - $\text{prop}_\sigma = \neg (\text{prop}_\sigma \lor (e_1 \land X_{e_2}(\ldots \land X_{e_k}(\ldots)))$
    - Define property $\phi_M$ corresponding to $M$ as
      - $\phi_M = \neg \bigvee_{\sigma \in \text{Lin}(M)} \text{prop}_\sigma$
    - A counter-example to $\phi_M$ is a test satisfying $M$. 
LTL Model Checking – does $M \models \varphi$

1. Consider $\neg \varphi$. None of the exec. traces of $M$ should satisfy $\neg \varphi$.
2. Construct a finite-state automata $A_{\neg \varphi}$ such that $\text{Language}(A_{\neg \varphi}) = \text{Traces satisfying } \neg \varphi$.
3. Construct the synch product $M \times A_{\neg \varphi}$.
4. Check whether any exec trace $\sigma$ of $M$ is an exec trace of the product $M \times A_{\neg \varphi}$ i.e. check $\text{Language}(M \times A_{\neg \varphi}) = \text{empty-set}$?
   - Yes: Violation of $\varphi$ found, report counterexample $\sigma$
   - No: Property $\varphi$ holds for all exec traces of $M$.

Recap: finite-state automata

- Regular languages:
  - Accept any finite-length string $\sigma \in \Sigma^*$ which ends in a final state.
- $\omega$-regular languages:
  - Accept any infinite-length string $\sigma \in \Sigma^\omega$ which visits a final state infinitely many times.
- Set of strings accepted = Language of the automata.

Finite automata

- Meaning as a regular language
  - $(a+b)^* b^*$
  - All finite length strings ending with $b$
- Meaning as a $\omega$-regular language
  - All infinite length strings with finitely many $a$

LTL properties to automata

- Given a LTL property $p$
  - we want to convert $p$ to an automata $A_p$ such that $\text{Language}(A_p) = \text{strings / traces satisfying } p$
- LTL properties are checked over infinite traces.
  - Given an infinite trace $\sigma$ and a LTL property $p$, we can check whether $\sigma \models p$
- To convert LTL properties to finite-state automata, consider automata accepting infinite length traces.
  - $\text{Language}(A_p)$ is $\omega$-regular, not regular.
LTL properties to automata

- Given a LTL property $\varphi$
  - We convert it to a $\omega$-regular automata $A_\varphi$.
- $\text{Language}(A_\varphi) = \{ \sigma \in \sum^\omega : \sigma \models \varphi \}$
  - $\text{Language}(A_\varphi)$ is defined as per the $\omega$-regular notion of string acceptance; it accepts infinite length strings.
  - All infinite length strings satisfying $\varphi$ form the language of $A_\varphi$.
  - Whether an infinite length string satisfies $\varphi$ (or not) is defined as per LTL semantics.

Recall: LTL Model Checking

1. Consider $\neg \varphi$. None of the exec. traces of $M$ should satisfy $\neg \varphi$.
2. Construct a finite-state automata $A_{\neg \varphi}$ such that
   - $\text{Language}(A_{\neg \varphi}) = \text{Traces satisfying } \neg \varphi$
3. Construct the synch product $M \times A_{\neg \varphi}$
4. Check whether any exec trace $\sigma$ of $M$ is an exec trace of the product $M \times A_{\neg \varphi}$ i.e. check $\text{Language}(M \times A_{\neg \varphi})$ = empty-set?
   - Yes: Violation of $\varphi$ found, report counterexample $\sigma$.
   - No: Property $\varphi$ holds for all exec traces of $M$.

Example: Verify $\text{GF} \varphi$

- Construct negation of the property $\neg \text{GF} \varphi = \text{FG} \neg \varphi$
- Construct automata accepting infinite length traces satisfying $\text{FG} \neg \varphi$

Product Automata Construction

- System Model $M$
- Property Automata $A_{\neg \varphi}$
- Product Automata $M \times A_{\neg \varphi}$

Product Automata

- (i) System Model $M$
- (ii) Property Automata $A_{\neg \varphi}$
- (iii) Product Automata $M \times A_{\neg \varphi}$
Recall: LTL Model Checking
1. Consider $\neg \phi$. None of the exec. traces of $M$ should satisfy $\neg \phi$.
2. Construct a finite-state automata $A_{\neg \phi}$ such that
   - $\text{Language}(A_{\neg \phi}) = \text{Traces satisfying } \neg \phi$
3. Construct the synch product $M \times A_{\neg \phi}$
4. Check whether any exec trace $\sigma$ of $M$ is an exec trace of the product $M \times A_{\neg \phi}$ i.e. check $\text{Language}(M \times A_{\neg \phi}) = \text{empty-set}$?
   - Yes: Violation of $\phi$ found, report counterexample $\sigma$
   - No: Property $\phi$ holds for all exec traces of $M$.

Emptiness Check
- Perform DFS from initial state until you reach an accepting state $s_{\text{acc}}$
- When you reach $s_{\text{acc}}$, remember $s_{\text{acc}}$ in a global var. and start a nested DFS from $s_{\text{acc}}$
- Stop the nested DFS if you can reach $s_{\text{acc}}$
- If no accepting cycles are found, report yes.
- If accepting cycles are found
  - Concatenate the two DFS stacks and report it as counterexample trace of the LTL property.
  - This algo. is implemented in SPIN model checker.

Nested DFS – step 1
- procedure dfs1(s)
  - push s to Stack1
  - add s to States1
  - if accepting(s) then
    - States2 := empty; seed := s; dfs2(s)
    - endif
  - for each transition $s \rightarrow s'$ do
    - if $s' \in$ States1 then dfs1(s')
    - endfor
  - pop s from Stack1
- end

Nested DFS – step 2
- procedure dfs2(s)
  - push s to Stack2
  - add s to States2
  - for each transition $s \rightarrow s'$ do
    - if $s' =$ seed then report acceptance cycle
    - else if $s' \notin$ States2 then dfs2(s')
    - endif
    - endfor
  - pop s from Stack2
- end